Line Distillation: A Mechanism to Improve Cache Utilization

Moinuddin K. Qureshi *David Thompson* *Thomas R. Puzak†* *Yale N. Patt*

High Performance Systems Group
Department of Electrical and Computer Engineering
The University of Texas at Austin
Austin, Texas 78712-0240

†IBM T. J. Watson Research Center, New York
trpuzak@us.ibm.com

TR-HPS-2006-002
16 February, 2006

This page is intentionally left blank

Line Distillation: A Mechanism to Improve Cache Utilization *

Moinuddin K. Qureshi, David Thompson, Thomas R. Puzak†, Yale N. Patt

Department of Electrical and Computer Engineering
The University of Texas at Austin
{moin,dave,patt}@hps.utexas.edu

†IBM T. J. Watson Research Center, New York
trpuzak@us.ibm.com

Abstract

Cache hierarchies play a very important role in bridging the speed gap between processors and memory. As this gap increases, it becomes increasingly important to intelligently design and manage a cache system. The performance of current caches is reduced because more than half of the data that is brought into the cache is never referenced, resulting in very low utilization. We propose *line distillation*, a technique to increase cache utilization by filtering the unused data from a subset of the lines and condensing the remaining useful data into smaller line-sizes. We describe three flavors of line distillation: *naive-distillation, static-K-distillation, and adaptive-distillation*. We also introduce the *distill cache*, a cache that supports line distillation and heterogeneous line-sizes. The line distillation technique reduces cache miss-rate by 21% on average.

1 Introduction

Caches exploit temporal and spatial locality that exists in memory reference streams. Temporal locality is exploited by keeping a copy of the data associated with a memory reference so that subsequent references to the same address can be satisfied by the cache. Spatial locality is exploited by caching more data than is necessary for a single memory reference in anticipation of future accesses to contiguous addresses. In this paper, we explore spatial locality as it affects cache design decisions.

There are three basic transactions that take place in a cache: *access*, *fill*, and *evict*. Cache accesses consist of loads and stores which take place between the cache and the processor. Access transactions take place at the *word-size* granularity as defined by the ISA which the machine is implementing and for which the cache is being designed. Line fills and evictions occur between the cache and the next level of the memory hierarchy and refer to placing data into and removing data from the cache, respectively. Fill and evict transactions take place at the *line-size* granularity as defined by the microarchitect designing the cache. The line-size must be at least as large as the word-size but is otherwise independent, and a typical line-size is 8-16 times the corresponding word-size in a machine. Using large line-sizes reduces the storage requirement for the cache's tag structure and provides a performance improvement proportional to

*This research was conducted when the first author was an intern at the IBM T.J. Watson Research Center during Summer 2004. A copy of this document was submitted on 5 August 2004 to the IBM Watson conference on Interaction between Architecture, Circuits, and Compiler (IBM internal session).

the amount of spatial locality present in the memory reference stream. If there is little spatial locality, however, the use of large line-sizes results in low utilization of the data stored in the cache. Low utilization increases the cache miss-rate and reduces system performance. Large line-sizes also require more bandwidth than small line-sizes due to the amount of data that must be transferred for fill and evict transactions. Small line-sizes, on the other hand, relax bandwidth requirements and increase cache-utilization but may result in lower overall cache performance if spatial locality is high. We refer to the presence of unused data in the cache as *cache pollution*.

A cache line that contains a significant amount of pollution is said to be *sparse*, whereas a line with very little pollution is said to be *dense*. We propose a mechanism to reduce the amount of pollution in a cache by extracting the useful data from sparse cache lines and discarding the data that is never accessed. The extracted subset of the cache line is compacted into a dense line. Both sparse and dense lines are kept in the same physical cache structure which we refer to as a *distill cache*. A first level distill cache yields a 21% reduction in miss-rate relative to a traditional baseline cache design.

2 Methodology

We use a trace driven cache simulator for our experiments. The baseline for all experiments is a 32KB, 4-way set-associative cache with a 128 byte line size, and the replacement policy is true LRU.

The traces used in this study were taken from both commercial and SPEC benchmarks. The commercial benchmarks OLTP, DB, and JAVA were compiled for a server machine. From the SPEC CPU2000 benchmark suite we selected the seven integer benchmarks that generate the greatest number of cache misses per 1000 instructions (MPKI). The SPEC benchmarks were compiled for the Alpha ISA and were run using the *test* input set. Table 1 lists a brief description of each benchmark as well as the total instruction count and the total number of data memory references (loads and stores) for the corresponding trace.

Table 1: Benchmark characteristics.

Benchmark	Description	Inst cnt.	Data ref.
OLTP	Online Transaction proc.	32 M	19 M
DB	Commercial Database app.	51 M	28 M
JAVA	JAVA workload	7 M	3.5 M
bzip2	File compression	418 M	138 M
crafty	Artificial intelligence	171 M	21 M
gzip	LZ77 coding	366 M	91 M
mcf	Vehicle scheduling	173 M	73 M
perlbmk	Scripting language	99 M	36 M
vortex	Object oriented database	165 M	71 M
vpr	Place and Route	567 M	222 M

3 Problem

We now describe cache pollution and its effects on cache performance, and we propose a technique to improve cache utilization by removing pollution on a per-line basis.

3.1 The Problem - Cache Pollution

In a traditional cache, the line-size is determined by the cache designer and remains permanently fixed after the machine is built. When the spatial locality of an application's memory reference stream is low, only a small part of each cache-line is used. Because the cache is unable to adapt its line-size to accommodate application behavior, a large amount of space in the cache is allocated to the storage of data that will never be accessed. In such a scenario, we say that the cache is *polluted*. We qualitatively define pollution as data that is placed into the cache which does not get accessed prior to being evicted. We use a simple technique to measure pollution in simulation. Each cache line is divided into a fixed number of uniformly sized sectors. When a line is evicted from the cache, we record the number of sectors that were accessed while the line was resident in the cache. Figure 1 shows the histogram of sectors touched for line-sizes of 64, 128, and 256 bytes. The bar labeled *amean* is the arithmetic mean across all ten benchmarks.

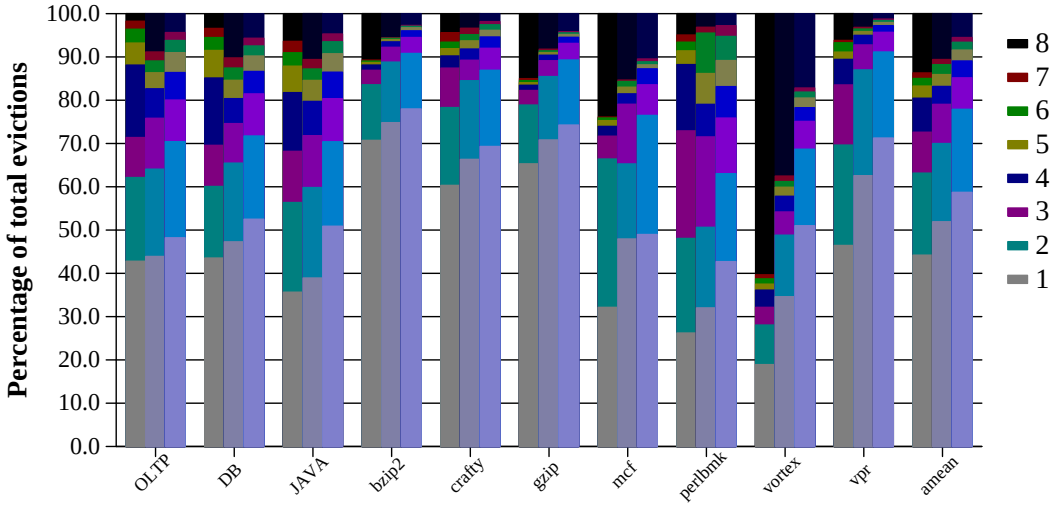

Figure 1: Distribution of number of sectors touched in a line before it gets evicted. The leftmost bar represents 64B line-size, the middle bar represents 128B line-size, and the rightmost bar represents 256B line-size.

A significant percentage of sectors in each cache line are never accessed. We introduce the term *density* to quantify the number of useful sectors in a cache line. The *density* of a cache line is defined as the number of sectors that are accessed while the line is resident in the cache. A quantitative measure of cache pollution is thus the difference between the total number of sectors in a line and the density of the line. Very dense cache lines have very little pollution. Therefore, to maximize cache utilization and to minimize cache pollution, a cache should contain mostly dense lines. Table 2 shows the average *density* for the baseline cache with 64, 128, and 256 byte line-sizes.

On average more than two-thirds of the data in the cache is pollution. This pollution results in increased cache miss-rate and decreased system performance. Next, we introduce a novel technique to reduce cache pollution.

Table 2: Density of a cache line.

Line-size	OLTP	DB	JAVA	bzip	crafty	gzip	mcf	perlbmk	vortex	vpr	**amean**
64B	2.48	2.59	2.86	2.04	2.03	2.37	3.29	2.85	5.69	2.33	**2.85**
128B	2.67	2.70	2.89	1.68	1.83	1.90	2.74	2.88	4.21	1.73	**2.5**
256B	2.35	2.31	2.32	1.50	1.65	1.62	2.37	2.54	2.82	1.50	**2.1**

3.2 The Solution - Line Distillation

We define the *footprint* of a cache line as a bit vector consisting of one bit for each sector in the line. When a line is placed into the cache, its footprint is reset to all zeros. When a sector is accessed, its bit in the footprint is set to 1, and cannot be reset while the line remains resident in the cache. The graph shown in Figure 1 was generated using an eight bit footprint. The replacement policy used in our experiments was true LRU. When a line is placed into the cache, it is marked as the most recently used (MRU) element in its set. As the program executes, the line travels back and forth between the various states of recency. If an access to the set containg the line results in a miss, and the line is the least recently used (LRU) element, it is evicted from the cache to make room for the new data.

We observe that the footprint of a cache line rarely changes if the line is no longer the MRU element in its set. Figure 2 shows the distribution of footprint updates in each state of a four-way LRU algorithm. Each cache line is counted only once with each line contributing to the count of the lowest LRU state in which a footprint update occured. For example, cache line A is in the MRU position when its first footprint update occurs. Later, line A moves to the second most recently used position. An access to a different sector causes the footprint to be updated and the line to move back into the MRU position. Line A is never accessed again and is eventually evicted from the cache. In this scenario, line A will contribute to the bucket labled “Position 1” in Figure 2. Each cache line may undergo at most eight footprint updates prior to eviction.

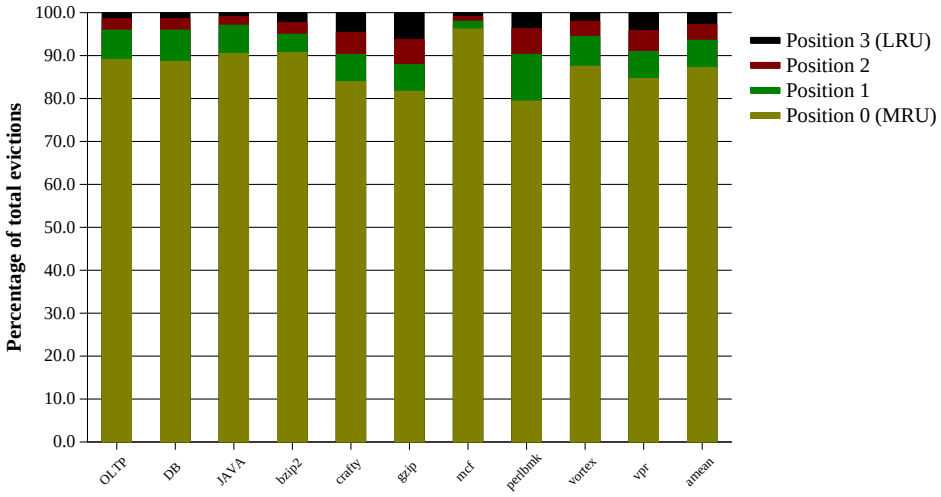

Figure 2: Distribution of footprint changes vs LRU position.

On average, 88% of changes to the footprint occur while a cache line is in the MRU position. Only 3% of the

changes to the footprint occur while a cache line is in the LRU position. This indicates that once a cache line reaches the LRU position in its set, its density has stabilized. By tracking footprints during run-time, the useful sectors of a cache line may be extracted and compacted into a dense line, and the sectors which cause pollution may be filtered out and discarded. We refer to this technique as *line distillation* and a cache that supports line distillation as a *distill cache*. The next section describes the design of a *distill cache* in detail.

4 Distill Cache

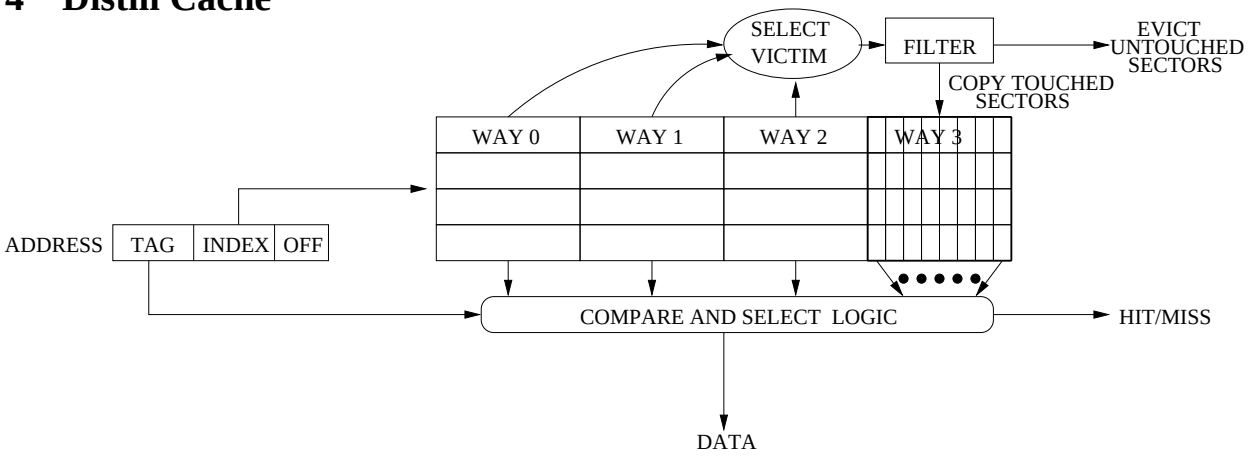

Figure 3: Distill Cache.

Figure 3 shows the organization of a four-way associative distill cache. Cache lines are divided into eight sectors for the purposes of monitoring density in the form of a footprint. One of the four ways (way 3) has the capacity to store up to eight sectors from eight different cache lines and is referred to as a *dense way*. A dense way contains additional tag-store entries to uniquely identify sectors belonging to different cache lines. In figure 3, way 3 contains eight tag-store entries. Each tag store entry in a dense way contains tag bits, status information (i.e. valid, LRU bits) and a sector tag which identifies the position of the sector in the original cache line. Ways 0-2 in the figure are referred to as *normal ways*, and each normal way contains a single tag-store entry. The tag-store entry for a normal way consists of tag bits, status information, and the current footprint for the line.

4.1 Operation

4.1.1 Cache Hit

For each cache access, the tag bits from both the normal ways and the dense way of the target set are compared to the referencing address. If a tag match is found in a normal way, the sector footprint is updated if necessary, and the cache signals a hit. If a tag match is found in the dense way, however, the sector bits of the matching tag must then be compared to the corresponding bits of the referencing address. We refer to this second comparison as a *sector comparison*. If the sector comparison results in a match, a cache hit is signaled and the data is provided from the sector.

Replacement information in the distill cache is maintained at two granularities. The normal ways maintain LRU information at the line-size granularity. The dense way maintains separate LRU information at the sector-size granularity.

4.1.2 Cache Miss

If a cache miss occurs, the LRU line from the set of normal ways is selected for replacement. This line is then distilled by extracting the sectors indicated by its footprint and copying the sectors into the dense way. To make room for the distilled line, the least recently used sectors in the dense way are evicted if there are not enough free sectors available. The originally requested line is then installed in the (now free) normal way. The footprint of the line is initialized with a single bit set corresponding to the specifically requested sector.

Every cache miss is classified as either a *classic miss* or a *hole miss*. A classic miss refers to a failed tag comparison in both the normal ways and the dense way. A hole miss refers to a successful tag comparison in the dense way followed by a failed sector comparison. When a hole miss occurs, one or more sectors of the original cache line is present in the dense way. These sectors are invalidated before the miss is serviced.

5 Results, Analysis, and Optimizations

In this section we discuss the management of the dense way through the comparison of three different distillation models.

5.1 Naive Distillation

In the simplest form of distillation, all of the sectors forming the cache line footprint are copied into the dense way. We refer to this as *naive distillation.* Table 3 shows the misses per 1000 instructions (MPKI) for the baseline cache, and a distill cache of the same size and configuration using naive distillation. Naive distillation results in an average MPKI reduction of 12.5%.

Table 3: Misses per 1000 instructions for baseline and baseline with naive distillation.

	OLTP	DB	JAVA	bzip	crafty	gzip	mcf	perlbmk	vortex	vpr	**amean**
Baseline	38.6	38.5	13.8	7.73	6.38	7.42	60.49	4.76	6.15	19.84	**20.4**
Naive dist	33.3	35.6	11.97	7.08	2.96	7.34	53.85	3.94	4.59	17.92	**17.85**
% Reduction	13.7%	7.7%	13.5%	8.4%	53.6%	1.08%	10.98%	17.23%	25.37%	9.68%	**12.5%**

5.2 Static-K Distillation

A problem that arises with naive distillation is the fact that a very dense line can replace all the sectors in the dense way, potentially evicting data from up to eight different cache lines. For very dense lines, it may be more beneficial to retain the contents of the dense way and simply evict the distilled line rather than store it in the dense way. The

decision to discard or cache the distilled line can be decided using a threshold. We define a *distillation threshold* K as an integer strictly less than the number of sectors in a line. Only evicted cache lines with a density less than or equal to K may be cached in the dense way. Lines with a density greater than K are simply discarded. We refer to this mechanism as *static-K distillation*. Table 4 shows the reduction in MPKI relative to the baseline cache as K is varied from 1 to 7. The case K = 8 is equivalent to naive distillation.

Table 4: Percent reduction in MPKI relative to the baseline for a distill cache using static-K distillation. The greatest improvement for each column is denoted using **bold.**

K	OLTP	DB	JAVA	bzip	crafty	gzip	mcf	perlbmk	vortex	vpr	amean
1	16.7%	11.4%	25.5%	9.2%	47.3%	-0.8%	**18.8%**	15.8%	27.6%	7.7%	16.4%
2	**20.6%**	**12.9%**	**25.8%**	**9.8%**	60.7%	0.5%	15.9%	**29.8%**	37.1%	**11.3%**	**18.0%**
3	19.9%	12.3%	22.6%	9.8%	**61.4%**	1.1%	14.0%	24.4%	**37.4%**	11.1%	16.9%
4	18.8%	11.4%	20.1%	9.7%	60.8%	1.3%	13.2%	23.5%	35.8%	11.0%	16.0%
5	17.9%	10.6%	18.2%	9.7%	59.7%	**1.5%**	12.8%	21.6%	34.5%	10.9%	15.4%
6	17.1%	9.9%	17.6%	9.6%	58.5%	1.5%	12.2%	19.1%	33.5%	10.8%	14.7%
7	16.4%	9.4%	16.6%	9.6%	56.9%	1.3%	11.9%	18.5%	31.9%	10.7%	14.2%

On average, K = 2 results in the greatest average reduction in cache misses, reducing misses by 18%. Different benchmarks prefer different distillation thresholds, and the best threshold may change during different phases of program execution.

5.3 Adaptive Distillation

The distillation threshold with which a benchmark achieves the lowest cache miss-rate is both unique to the benchmark and correlated to the average density of the benchmark (see table 2). We refer to a mechanism that determines the distillation threshold at run-time as *adaptive distillation*. Using an execution interval of 100,000 cache accesses, we compute the average density of evicted cache lines across an interval. This average density becomes the new distillation threshold for the next execution interval. We use two registers to compute average density: an *eviction-count* register and a *sector-count* register. The eviction-count register is incremented for each line that is evicted from a normal way. The sector-count register is incremented by the size of the footprint of the evicted line. The footprint size is a one's count of the footprint bit vector. At each interval boundary, the average density is computed using integer division, dividing the sector-count register by the eviction-count register. The two registers are then cleared for the next interval. Table 5 summarizes the results for adaptive distillation. Adaptive distillation reduces MPKI by an average of 21.9%.

Table 5: Percent reduction in MPKI relative to the baseline for a distill cache using adaptive distillation.

OLTP	DB	JAVA	bzip	crafty	gzip	mcf	perlbmk	vortex	vpr	amean
20.9%	15.4%	36.5%	9.6%	60.7%	-0.5%	17.3%	29.8%	37.1%	8.2%	**21.9%**

6 Related Work

Sectored caches provide a low cost solution for tag directories. Instead of a cache line, only a portion of a cache line, i.e. a sector, is fetched on a miss. Sectored caches utilize bandwidth efficiently by only fetching the requested sector. However, this solution comes at the expense of inefficiently using the cache space as some sectors remain invalid.

Gonzalez et al. [1] proposed a cache with multiple caching strategies for numerical codes. Their design consists of a spatial cache, a temporal cache, and a predictor. Depending on the prediction, the fetched line is either placed in the spatial cache or the temporal cache. The proposed scheme works well for numerical codes containing regular stride based access patterns, but it does not scale well to integer codes.

Johnson [2] predicts spatial locality of access over a region of memory and varies fetch size on a miss. Kumar et al [3] use a spatial footprint predictor to predict the spatial footprint of a cache line. Depending on the prediction the number of lines fetched on a miss is varied. Both of the above schemes use a uniform line-size cache and the fetch pattern is varied depending on the outcome of a predictor. These schemes require the area overhead of a hardware predictor. Moreover, the tag store either contains a large number of tag store entries [2] or implements a constrained allocation policy [3].

7 Conclusion

When the memory reference stream of an application exhibits low spatial locality, cache utilization decreases due to significant levels of cache pollution. Much of the data stored in the cache is never used before being evicted. In this paper, we propose the *line distillation* technique to filter and remove the polluting data in a cache, resulting in increased cache utilization and fewer cache misses. We propose three distillation mechanisms: naive distillation, static-K distillation, and adaptive distillation. We describe the design of cache with heterogeneous line-sizes that supports the line distillation technique. A cache using the adaptive distillation technique has an average of 22% fewer cache misses per 1000 instructions than a traditional set associative cache. While this paper describes several aspects of line distillation, there are several design issues to be further explored. Examples include the performance impact of multiple dense ways and replacement policy heuristics regarding the eviction of sectors in the dense way. We plan to continue the exploration of these issues in future work.

Acknowledgments

We thank Alan Hartstein and Philip Emma for discussion and feedback on this work.

References

[1] A. Gonzalez, C. Aliagas, and M. Valero. A data cache with multiple caching strategies tuned to different types of locality. In *Proceedings of the 1995 International Conference on Supercomputing*, pages 338–347, 1995.

[2] T. L. Johnson. *Run-time adaptive cache management*. PhD thesis, University of Illinois, Urbana, IL, May 1998.

[3] S. Kumar and C. Wilkerson. Exploiting spatial locality in data caches using spatial footprints. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 357–368, 1998.